Chapter 1: Automation with Other Office Programs

In This Chapter

- **Getting up to speed on Automation**
- **Working with Object Libraries**
- **Adding a contact to Outlook**
- **Merging data with a Word document**
- **Exporting data to Excel**

In Book VIII, we show you Visual Basic for Applications (VBA) and present some of the wonderful ways you can take control of your Access database. You can use VBA to open and close forms, print reports, loop through tables and change data, and modify form properties.

Well, VBA isn't there just for Access. You can also use VBA to control other Microsoft Office applications, including Outlook, Excel, Word, and PowerPoint. With VBA, the possibilities are virtually endless when you consider what some advanced users do in these Office applications on a daily basis. This chapter explains Automation and gives several examples of how Access can interact with other Office programs.

Understanding Automation

Automation (with a lowercase *a)* came about during the Industrial Revolution to replace tasks performed by humans with faster, more efficient methods. Phone operators no longer plug and unplug wires manually to make connections; large systems handle connections automatically. People don't do all the work of assembling cars on assembly lines; industrial robots now handle the bulk of the duties. We humans just get in the way.

In the world of VBA, *Automation* (with a capital *A)* refers to the capability of a program to *expose* itself (make itself available) to VBA so that VBA can control it behind the scenes, with little or no human interaction. Humans just slow the process anyway. Other programming languages (such as VB.NET, C++, and C#) use Automation as well, but because VBA is the language of Access, we focus in this chapter on using VBA.

Automation with other Microsoft Office programs works only when you have those programs installed on your computer. If you don't have Word, Excel, or Outlook installed, you won't be able to control them from Access.

Working with Object Libraries

To use VBA to control another program, you need to have access to that program's object library. Each program has its own properties and methods that allow VBA to control it. Just as each object (such as a form, text box, or button) has its own properties and methods, each application — including Access — has a set of properties and methods, collectively referred to as the *object library.*

To access another program's object library, you first have to tell VBA where to find it. To add an object library to your VBA project, choose Tools➪References in Visual Basic Editor and then add the desired object libraries, as shown in Figure 1-1.

For more information on using Visual Basic Editor, see Book VIII, Chapter 1.

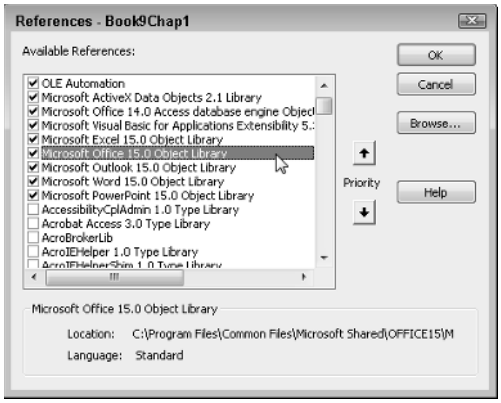

Figure 1-1: Choose object libraries from the References dialog box.

In Figure 1-1, we chose Microsoft Excel 15.0 Object Library, Microsoft Office 15.0 Object Library, Microsoft Outlook 15.0 Object Library, Microsoft Word 15.0 Object Library, and Microsoft PowerPoint 15.0 Object Library. Selected items appear at the top of the list when you open the References dialog box, so they may not always appear in the same order.

If you have multiple versions of a program installed on your computer (Excel 2007 and Excel 2013, for example), you see different versions of the Excel object library in the References dialog box (refer to Figure 1-1). If you're sure that you'll be working only in the latest version, choose the version with the highest number. Applications in Office 2013 are version 15.0, whereas applications in Office 2010 are version 14.0.

Exploring object libraries

After adding a reference to a program's object model, you can explore that program's objects, properties, and methods through the Object Browser. To open the Object Browser from Visual Basic Editor, choose View➪Object Browser or press F2. When you open the Object Browser, it shows the objects for everything VBA has access to. To limit the list to a specific library, choose the library's name from the Project/Library drop-down menu in the top-left corner of the Object Browser window. In Figure 1-2, we chose Excel to show only the classes and members related to Microsoft Excel, and then chose Application from the list of Classes to display the members of the Application class.

For more information on object models and object libraries, see Book VIII, Chapter 1.

Figure 1-2: Use the Object Browser to view a program's object model.

Each application exposes a lot of objects to VBA — way too many for you (or any sane person) to remember. We don't have enough room in this book to define every property and every method for each Office application. We'd probably need a book just for each application, which wouldn't make too many trees very happy, would it? Instead, you have to be able to get the information you need when you need it.

To find out more about a selected object, property, or method in the Object Browser, click the Help icon — the yellow question mark — in the Object Browser window (refer to Figure 1-2).

Using the Application object

Each application exposes its own set of objects to VBA, but one object that all applications have in common is `Application`. The `Application` object exposes a program's objects, properties, and methods to VBA. When a program is open, its objects are available to VBA. If VBA opens a Word document, for example, everything in that Word document is also exposed. VBA can do anything in the Word document that a human can do from the Word Ribbon.

To take control of an application, you first have to create an instance of the application in VBA. Creating an *instance* is basically the same as opening the program from the Windows Start menu. When you start Microsoft PowerPoint on your computer, for example, you're creating an instance of PowerPoint on your computer.

To create an instance of an application in VBA, you have to declare a variable that references that object. The variable can have any name you like, but you should attempt to give it a meaningful name. The syntax for declaring an object variable is

```
Dim objectVariable as New program.Application
```

For more information on declaring and using variables, see Book VIII, Chapter 3.

The *objectVariable* in the preceding example is the name of the variable. The *program* is a reference to one of the Office applications (such as Word, Excel, or Outlook). The `Application` part refers to the `Application` object of that program. The `New` keyword ensures that VBA creates a new instance of the program. Here are some examples of declaring new instances of the Office programs:

```
Dim XL as New Excel.Application
Dim Wrd as New Word.Application
Dim Olk as New Outlook.Application
Dim PPT as New PowerPoint.Application
```

After you declare the object variable for the desired program, you can control that program. To take control of the program, you must open the program from VBA. The syntax for opening a program in VBA is

```
Set objectVariable as CreateObject("program.Application")
```

where the *objectVariable* is the same name you specified in the `Dim` statement and *program* is the name of the application. When you use the `Dim` statements described in this section, the corresponding `Set` statements for opening the applications are

```
Set XL as CreateObject("Excel.Application")
Set Wrd as CreateObject("Word.Application")
Set Olk as CreateObject("Outlook.Application")
Set PPT as CreateObject("PowerPoint.Application")
```

To control another program with VBA, you must first add the program's object library to VBA, using the References dialog box (refer to "Working with Object Libraries," earlier in this chapter).

In the next few sections, you see how Access can share information with Outlook, Word, and Excel — oh, my!

Adding Contacts to Outlook

Suppose that you're working on an Access program, and your users suggest that it would be a good idea to give them the capability to add contacts from the Access database to Microsoft Outlook. You could be mean and tell them to type the contacts themselves, or you can impress them by adding a button to a form that lets them add the current contact to their Outlook contacts.

Adding the contact button and code

Consider the form shown in Figure 1-3. This form is a basic customer contact form that you might find in any of your applications, with one exception: the addition of an Add to Outlook Contacts button.

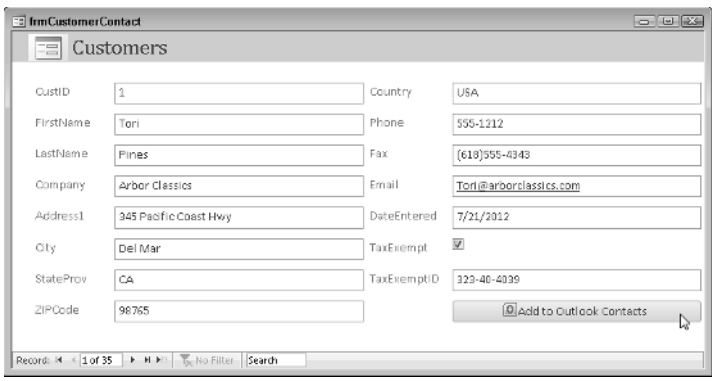

Figure 1-3: Changing an Access form to let users add contacts to Outlook.

Now, just adding the button doesn't accomplish much; you also have to add code to the button's `Click` event procedure. In the form's Design view, double-click the button to show the property sheet, click the Event tab, click the ellipsis button to the right of the `On Click` event property, and then click Code Builder to open Visual Basic Editor to the button's `Click` event procedure. The code looks something like this:

```
Private Sub cmdOutlook_Click()

  'Open Instance of Microsoft Outlook
  Dim Olk As Outlook.Application
  Set Olk = CreateObject("Outlook.Application")

  'Create Object for an Outlook Contact
  Dim OlkContact As Outlook.ContactItem
  Set OlkContact = Olk.CreateItem(olContactItem)

  'Set Contact Properties and Save
  With OlkContact
    .FirstName = Me.FirstName
    .LastName = Me.LastName
     Me.Email.SetFocus
    .Email1Address = Me.Email.Text
    .CompanyName = Me.Company
    .BusinessAddressStreet = Me.Address1
    .BusinessAddressCity = Me.City
    .BusinessAddressState = Me.StateProv
    .BusinessAddressPostalCode = Me.ZIPCode
    .BusinessTelephoneNumber = Me.Phone
    .BusinessFaxNumber = Me.Fax
    .Save
  End With

  'Let User know contact was added
  MsgBox "Contact Added to Outlook."

  'Clean up object variables
  Set OlkContact = Nothing
  Set Olk = Nothing

End Sub
```

This example may look like a lot of code, but it's really just a series of small steps. In layman's terms, this procedure creates and sets a variable for the Outlook application, creates and sets a variable for an Outlook contact, sets the properties of the Outlook contact object to values from the form, saves the Outlook contact, displays a message box, and cleans up the object variables. In the following section, we give you a detailed look at this example.

Examining the contact-form code

The first two statements below the first comment declare an object variable named `Olk` and set it to an open instance of Microsoft Outlook:

```
'Open Instance of Microsoft Outlook
Dim Olk As Outlook.Application
Set Olk = CreateObject("Outlook.Application")
```

The `Application` object for Outlook lets you create items within Outlook, just as though you'd opened Outlook and navigated through the program. The second comment and the next two lines are as follows:

```
'Create Object for an Outlook Contact
Dim OlkContact As Outlook.ContactItem
Set OlkContact = Olk.CreateItem(olContactItem)
```

These lines declare an object variable named `OlkContact` and create that contact by using the `CreateItem` method of the Outlook Application object. This code is VBA's way of clicking Contacts and then clicking the Click Here to Add a New Contact line at the top of the Outlook window.

Now look at the next block of code:

```
'Set Contact Properties and Save
With OlkContact
   .FirstName = Me.FirstName
   .LastName = Me.LastName
    Me.Email.SetFocus
   .Email1Address = Me.Email.Text
   .CompanyName = Me.Company
   .BusinessAddressStreet = Me.Address1
   .BusinessAddressCity = Me.City
   .BusinessAddressState = Me.StateProv
   .BusinessAddressPostalCode = Me.ZIPCode
   .BusinessTelephoneNumber = Me.Phone
   .BusinessFaxNumber = Me.Fax
   .Save
End With
```

The `With...End With` block of code sets the properties of the Outlook `ContactItem`. The `ContactItem` object has many properties that you can see via the Object Browser. This example uses only a few of these properties and sets them to the values from the form. Everything that uses the `Me` keyword reads a value from the form shown in Figure 1-3, earlier in this chapter.

The last few statements tell the user that the contact was added and reset the object variables.

The one tricky part of the code lies in how Access stores an e-mail address. Access doesn't just store the e-mail address as text; it stores additional information along with the e-address. So when VBA finds the e-mail address on the form, you want it to read only the text component of the e-mail address field. To direct VBA to read this property on the form, you must use the `SetFocus` method of the Email text box to make sure that the control *has the focus* — that is, the cursor is in that field.

The `Save` method of the `ContactItem` object saves the contact in Outlook. The remaining code displays a message to let you know that the contact was added to Outlook; then it cleans up the variables before the `Click` event procedure ends. Figure 1-4 shows the contact in Outlook.

The code in this section and throughout this chapter is designed to show you how to use the object libraries of other Office components. Error messages may appear for various reasons, such as null values or typos. For more information on handling errors in VBA, see Book VIII, Chapter 6.

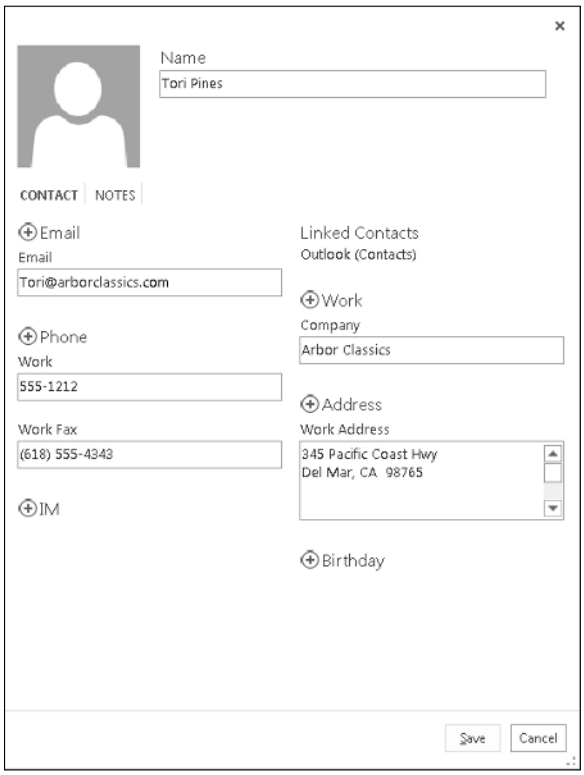

Figure 1-4: A contact added to Outlook from VBA.

Outlook's object library exposes many more objects and methods than we describe here. You can do anything from VBA that you can do from the Outlook program, such as compose and send e-mail messages, create and schedule calendar items, and build tasks and to-do lists. To find out more about these objects, properties, and methods, open the Object Browser, and choose Outlook from the Project/Library drop-down menu in the top-left corner.

Merging Data with a Word Document

Microsoft Word is probably the most widely used word-processing program in the world, if not the universe. If you have Microsoft Office installed, you almost certainly have Word installed as well. Many people in any given work environment know how to use and edit Word documents, but they may not know how to create and modify an Access report. Using Automation, you can enable some users to edit the body of a form letter in Word and then print that letter from Access. This section shows you how.

To control Word 2013 with VBA, you must first add the Microsoft Office Word 15.0 Object Library to VBA by using the References dialog box (see "Working with Object Libraries," earlier in this chapter).

Creating a Word template

To put data from Access in a Word document, you have to tell Access where in the Word document to put that data. One method is to create bookmarks in the Word document; later, Access can replace these bookmarks with data from the database. A bookmark in Word is just a placeholder. If you do this in a Word template file (`.dotx` file), you can easily create new documents based on this template.

First, use Word to create a new document, and format the document however you want. You can add a company logo and other letterhead information, and type the body of the letter. This task should be pretty easy. When you save the file, choose File➪Save As to open the Save dialog box, and from the Save As Type drop-down menu, choose Word Template (*.dotx).

Viewing and inserting bookmarks

When you get the document ready, you need to add bookmarks where you want the data from Access to go. Bookmarks in Word are usually hidden, so you need to start by displaying them so that you can see what you're doing. Follow these steps:

1. **In Microsoft Word, choose File➪Options.**
2. **Click the Advanced option on the left side of the Word Options window.**

3. **Scroll down to the Show Document Content section, and select the Show Bookmarks option.**
4. **Click OK.**

   Now you can insert bookmarks into your Word template.
5. **Move the cursor to where you want the bookmark to appear in the Word document.**
6. **Type a short, meaningful name for the bookmark.**

   The name can't contain spaces or punctuation, and it must start with a letter.
7. **Select the text by double-clicking the name you just typed, and copy it to the clipboard by pressing Ctrl+C.**
8. **On the Word Ribbon, click the Insert tab and then click Bookmark in the Links group.**

   The Bookmark dialog box appears, as shown in Figure 1-5.

Figure 1-5: Adding bookmarks to a Word template (`.dotx`) file.

9. **Paste the typed name into the Bookmark Name field by pressing Ctrl+V.**
10. **Click the Add button to create the bookmark.**

    Square brackets appear around the text to indicate the bookmark.

For this example, create three bookmarks — `CurrentDate`, `AccessAddress`, and `AccessSalutation` — and then save the `.dotx` file in the same folder as your database.

Adding the merge button

Now that the Word template is ready to go, create another button on the Customer form that sends data from the form to Word. You might call it Send Welcome Letter, as shown in Figure 1-6.

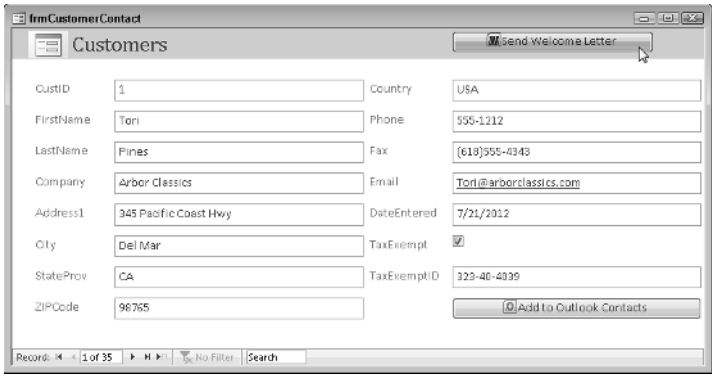

Figure 1-6: Adding another button to send data to Word.

Entering the merge code

When the button is on the form, you can add the following code to this button's `Click` event procedure to send the data to the Word document:

```
Private Sub cmdWord_Click()

  'Declare Variables
  Dim sAccessAddress As String
  Dim sAccessSalutation As String

  'Build sAccessAddress
  sAccessAddress = FirstName & " " & LastName & _
    vbCrLf & Company & vbCrLf & Address1 & _
    vbCrLf & City & ", " & StateProv & "  " & ZIPCode

  'Build sAccessSalutation
  sAccessSalutation = FirstName & " " & LastName

  'Declare and set Word object variables
  Dim Wrd As New Word.Application
  Set Wrd = CreateObject("Word.Application")

  'Specify Path to Template
  Dim sMergeDoc As String
  sMergeDoc = Application.CurrentProject.Path & _
     "\WordMergeDocument.dotx"

  'Open Word using template and make Word visible
```

```
  Wrd.Documents.Add sMergeDoc
  Wrd.Visible = True

  'Replace Bookmarks with Values
  With Wrd.ActiveDocument.Bookmarks
    .Item("CurrentDate").Range.Text = Date
    .Item("AccessAddress").Range.Text = sAccessAddress
    .Item("AccessSalutation").Range.Text = sAccessSalutation

  End With

  'Open in Print Preview mode, let user print
  Wrd.ActiveDocument.PrintPreview

  'Clean Up code
  Set Wrd = Nothing

End Sub
```

Again, this example has a lot of code, but that code breaks down into several sections. It declares the variables you're going to use, sets the address and salutation variables, opens Word (using the template you created), replaces the bookmarks with values from Access, and shows the Print Preview view for the document. The following section takes a closer look at some key components of this code.

Examining the merge code

After declaring the string values, you set the `sAccessAddress` variable to a concatenated string of values from the form. You use the line-continuation character (an underscore) as well as the `vbCrLf` keyword, which starts a new line in the string variable, as follows:

```
  'Build sAccessAddress
  sAccessAddress = FirstName & " " & LastName & _
    vbCrLf & Company & vbCrLf & Address1 & _
    vbCrLf & City & ", " & StateProv & "  " & ZIPCode
```

You also build the `sAccessSalutation` variable by combining the first-name and last-name fields from the form, with a space in between:

```
  'Build sAccessSalutation
  sAccessSalutation = FirstName & " " & LastName
```

Next, use the syntax described in "Using the Application object," earlier in this chapter, to open an instance of Microsoft Word. Here's the code:

```
'Declare and set Word object variables
Dim Wrd As New Word.Application
Set Wrd = CreateObject("Word.Application")
```

After opening an instance of Word, set the location of the Word template that created earlier (see "Creating a Word template," earlier in this chapter). You use the `Path` property of `Application.CurrentProject` to get the location and then concatenate it with the filename. For this example, we named the Word template file `WordMergeDocument.dotx`, as follows:

```
'Specify Path to Template
Dim sMergeDoc As String
sMergeDoc = Application.CurrentProject.Path & _
   "\WordMergeDocument.dotx"
```

This code assumes that you saved the Word template file in the same folder as the Access database.

Next, use the `Add` method of the `Application.Documents` object to create a new document based on the template file. After creating a new Word document, set the `Visible` property of the `Wrd` object to true, letting the user see Word. When you open Word from VBA, it's invisible to the user unless you specify otherwise, as follows:

```
'Open Word using template and make Word visible
Wrd.Documents.Add sMergeDoc
Wrd.Visible = True
```

After viewing the document, use the `Bookmarks` collection within the `ActiveDocument` to add the values from Access. Replace the `CurrentDate` bookmark with the system date, using the `Date()` function. Then replace the `AccessAddress` and `AccessSalutation` bookmarks with the variables set earlier in the code, as follows:

```
'Replace Bookmarks with Values
With Wrd.ActiveDocument.Bookmarks
  .Item("CurrentDate").Range.Text = Date
  .Item("AccessAddress").Range.Text = sAccessAddress
  .Item("AccessSalutation").Range.Text = sAccessSalutation
End With
```

Finally, switch to Print Preview view, and clean up the code in Word. Figure 1-7 shows the document with the bookmarks replaced by data from Access.

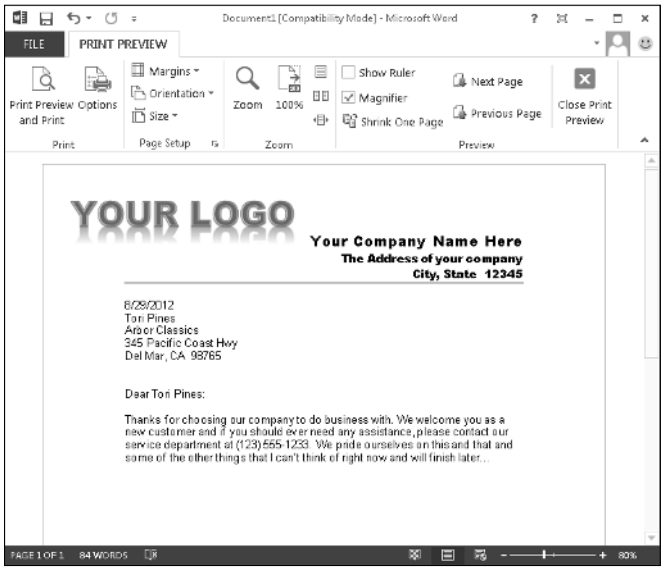

Figure 1-7: The final merged document in Word.

Just like the Outlook object library, the Word object library contains many more properties and methods that allow you to control Word as though you were clicking the various Ribbon commands and typing in the Word window. For assistance on all of these commands, use the Object Browser (choose View➪Object Browser or press F2 in Visual Basic Editor).

Exporting Data to Excel

Many Office users who are familiar with Excel just don't understand the power and flexibility of Access, and many executives are used to viewing and printing tables of data from an Excel spreadsheet. So even though you're convinced that everyone in your organization (and, perhaps, the world) should use Access instead of Excel, you'll still come across quite a few people who'd rather see the data in Excel than open an Access database.

Sure, you can export data to Excel (or other formats) from the Export group on the External Data tab, but that task requires one of those pesky humans to know what to do. As a compromise, you can automate the process by writing VBA code to export the data — and do several formatting tasks as well.

Adding the export button and code

Pretend that you're going to create a spreadsheet of phone numbers in the Customer table. You also want to add a meaningful title that includes the

date when the phone numbers were exported. You can create a button anywhere in your application to do this, so we'll just show you the code:

```
'Declare and set the Connection object
Dim cnn As ADODB.Connection
Set cnn = CurrentProject.Connection

'Declare and set the Recordset object
Dim rs As New ADODB.Recordset
rs.ActiveConnection = cnn

'Declare and set the SQL Statement to Export
Dim sSQL As String
sSQL = "SELECT FirstName, LastName, Phone FROM Customers"

'Open the Recordset
rs.Open sSQL

'Set up Excel Variables
Dim Xl As New Excel.Application
Dim Xlbook As Excel.Workbook
Dim Xlsheet As Excel.Worksheet

Set Xl = CreateObject("Excel.Application")
Set Xlbook = Xl.Workbooks.Add
Set Xlsheet = Xlbook.Worksheets(1)

'Set Values in Worksheet
Xlsheet.Name = "Phone List"
With Xlsheet.Range("A1")
  .Value = "Phone List " & Date
  .Font.Size = 14
  .Font.Bold = True
  .Font.Color = vbBlue
End With

'Copy Recordset to Worksheet Cell A3
Xlsheet.Range("A3").CopyFromRecordset rs

'Make Excel window visible
Xl.Visible = True

'Clean Up Variables
rs.Close
Set cnn = Nothing
Set rs = Nothing
Set Xlsheet = Nothing
Set Xlbook = Nothing
Set Xl = Nothing
```

As you can see, the code is starting to grow. It's not out of control, but procedures commonly grow to pages in length. Don't be afraid; as long as you break code into small chunks, it's not so hard to understand. The following section gives you the breakdown for this code.

The code in this section requires the ActiveX Data Objects Library to be selected in the References dialog box. For more information on ActiveX, see Book VIII, Chapter 5.

Examining the export code

The first chunk of code sets up the `Recordset` object with a simple SQL `Select` statement that gets the first name, last name, and phone number from the `Customers` table, as follows:

```
'Declare and set the Connection object
Dim cnn As ADODB.Connection
Set cnn = CurrentProject.Connection

'Declare and set the Recordset object
Dim rs As New ADODB.Recordset
rs.ActiveConnection = cnn

'Declare and set the SQL Statement to Export
Dim sSQL As String
sSQL = "SELECT FirstName, LastName, Phone FROM Customers"

'Open the Recordset
rs.Open sSQL
```

For more information on using recordsets and creating SQL statements in VBA code, see Book VIII, Chapter 5.

The next chunk of VBA code initializes the Excel objects so that you can manipulate them. You have three objects to declare when you're working with Excel: `Application`, `Workbook`, and `Worksheet`. By default, when you open Excel from the Start menu, the program opens to a new workbook, and each workbook contains at least one worksheet. Here's the code:

```
'Set up Excel Variables
Dim Xl As New Excel.Application
Dim Xlbook As Excel.Workbook
Dim Xlsheet As Excel.Worksheet

Set Xl = CreateObject("Excel.Application")
```

After opening the Excel `Application` object, use the `Add` method of the `Workbooks` collection to create a new workbook stored in the `Xlbook` variable, as follows:

```
Set Xlbook = Xl.Workbooks.Add
```

After adding a new `Workbook` to the Excel `Application`, set the `Xlsheet` variable to the first `Worksheet` of the `Workbook` object, using the `Worksheets` collection:

```
Set Xlsheet = Xlbook.Worksheets(1)
```

Now that the worksheet is initialized and set, it's time to start playing around. First, set the `Name` of the worksheet to something other than `Sheet1`. Then change cell `A1` to a meaningful heading that includes the date, and format the cell to use a larger, bolder, more colorful font, as follows:

```
'Set Values in Worksheet
Xlsheet.Name = "Phone List"
With Xlsheet.Range("A1")
  .Value = "Phone List " & Date
  .Font.Size = 14
  .Font.Bold = True
  .Font.Color = vbBlue
End With
```

Now it's time to take the data from the `Recordset` object and put it into the spreadsheet. There's no looping through the recordset here; just use the `CopyFromRecordset` method to copy the contents of a recordset into a particular range of cell. For this example, copy the data from the recordset starting with cell `A3`, as follows:

```
'Copy Recordset to Worksheet Cell A3
Xlsheet.Range("A3").CopyFromRecordset rs
```

Finally, make the Excel application visible so that you can see your data in Excel (as shown in Figure 1-8), and clean up the variables.

You can add much more code to this routine to fully customize the look of the spreadsheet. You can change the column widths, change the cells' background colors, and sort the data.

If you can perform a task with the mouse and keyboard in the Excel window, you can find a way to do it with VBA.

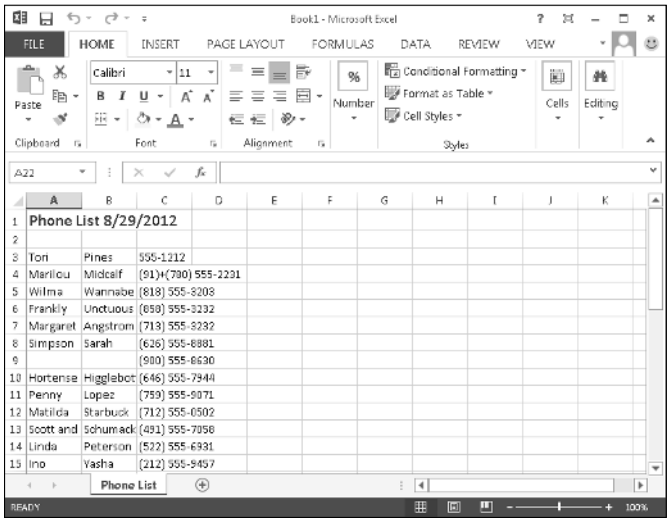

Figure 1-8: The Excel worksheet with customer phone numbers.

Automating Access with other Office programs can seem overwhelming at first, but when you know where to find help and examples, you'll be well on your way to beefing up your Access applications — and relying less and less on other humans to perform these tasks.